Semi-Automated Reasoning About Non-Determinism in C Expressions

Dan Frumin[1], Léon Gondelman[1], and Robbert Krebbers[2]

[1] Radboud University
{dfrumin,lgg}@cs.ru.nl
[2] Delft University of Technology
mail@robbertkrebbers.nl

Abstract. Research into C verification often ignores that the C standard leaves the evaluation order of expressions unspecified, and assigns undefined behavior to write-write or read-write conflicts in subexpressions—so called "sequence point violations". These aspects should be accounted for in verification because C compilers exploit them.
We present a verification condition generator (vcgen) that enables one to semi-automatically prove the absence of undefined behavior in a given C program for *any* evaluation order. The key novelty of our approach is a symbolic execution algorithm that computes a *frame* at the same time as a *postcondition*. The frame is used to automatically determine how resources should be distributed among subexpressions.
We prove correctness of our vcgen with respect to a new monadic definitional semantics of a subset of C. This semantics is modular and gives a concise account of non-determinism in C.
We have implemented our vcgen as a tactic in the Coq interactive theorem prover, and have proved correctness of it using a separation logic for the new monadic definitional semantics of a subset of C.

1 Introduction

The ISO C standard [22]—the official specification of the C language—leaves many parts of the language semantics either *unspecified* (*e.g.,* the order of evaluation of expressions), or *undefined* (*e.g.,* dereferencing a NULL pointer or integer overflow). In case of undefined behavior a program may do literally anything, *e.g.,* it may crash, or it may produce an arbitrary result and side-effects. Therefore, to establish the correctness of a C program, one needs to ensure that the program has no undefined behavior for *all* possible choices of non-determinism due to unspecified behavior.

In this paper we focus on the undefined and unspecified behaviors related to C's expression semantics, which have been ignored by most existing verification tools, but are crucial for establishing the correctness of realistic C programs. The C standard does not require subexpressions to be evaluated in a specific order (*e.g.,* from left to right), but rather allows them to be evaluated in *any* order. Moreover, an expression has undefined behavior when there is a conflicting write-write or read-write access to the same location between two *sequence points* [22,

6.5p2] (so called "sequence point violation"). Sequence points occur *e.g.,* at the end of a full expression (`;`), before and after each function call, and after the first operand of a conditional expression (`- ? - : -`) has been evaluated [22, Annex C]. Let us illustrate this by means of the following example:

```
int main() {
  int x; int y = (x = 3) + (x = 4);
  printf("%d␣%d\n", x, y);
}
```

Due to the unspecified evaluation order, one would naively expect this program to print either "`3 7`" or "`4 7`", depending on which assignment to `x` was evaluated first. But this program exhibits undefined behavior due to a sequence point violation: there are two conflicting writes to the variable `x`. Indeed, when compiled with GCC (version 8.2.0), the program in fact prints "`4 8`", which does not correspond to the expected results of any of the evaluation orders.

One may expect that these programs can be easily ruled out statically using some form of static analysis, but this is not the case. Contrary to the simple program above, one can access the values of arbitrary pointers, making it impossible to statically establish the absence of write-write or read-write conflicts. Besides, one should not merely establish the absence of undefined behavior due to conflicting accesses to the same locations, but one should also establish that there are no other forms of undefined behavior (*e.g.,* that no NULL pointers are dereferenced) for *any evaluation order*.

To deal with this issue, Krebbers [29,30] developed a program logic based on Concurrent Separation Logic (CSL) [46] for establishing the absence of undefined behavior in C programs in the presence of non-determinism. To get an impression of how his logic works, let us consider the rule for the addition operator:

$$\frac{\{P_1\}\,\mathtt{e}_1\,\{\Psi_1\} \qquad \{P_2\}\,\mathtt{e}_2\,\{\Psi_2\} \qquad \forall \mathtt{v}_1\,\mathtt{v}_2.\,\Psi_1\;\mathtt{v}_1 * \Psi_2\;\mathtt{v}_2 \vdash \Phi\;(\mathtt{v}_1 + \mathtt{v}_2)}{\{P_1 * P_2\}\,\mathtt{e}_1 + \mathtt{e}_2\,\{\Phi\}}$$

This rule is much like the rule for parallel composition in CSL—the precondition should be separated into two parts P_1 and P_2 describing the resources needed for proving the Hoare triples of both operands. Crucially, since P_1 and P_2 describe disjoint resources as expressed by the *separating conjunction* $*$, it is guaranteed that $\mathtt{e}_1$ and $\mathtt{e}_2$ do not interfere with each other, and hence cannot cause sequence point violations. The purpose of the rule's last premise is to ensure that for all possible return values $\mathtt{v}_1$ and $\mathtt{v}_2$, the postconditions Ψ_1 and Ψ_2 of both operands can be combined into the postcondition Φ of the whole expression.

Krebbers's logic [29,30] has some limitations that impact its usability:

- The rules are not algorithmic, and hence it is not clear how they could be implemented as part of an automated or interactive tool.
- It is difficult to extend the logic with new features. Soundness was proven with respect to a monolithic and ad-hoc model of separation logic.

In this paper we address both of these problems.

We present a new algorithm for symbolic execution in separation logic. Contrary to ordinary symbolic execution in separation logic [5], our symbolic executor takes an expression and a precondition as its input, and computes not only the postcondition, but also simultaneously computes a *frame* that describes the resources that have *not* been used to prove the postcondition. The frame is used to infer the pre- and postconditions of adjacent subexpressions. For example, in $\mathsf{e}_1 + \mathsf{e}_2$, we use the frame of e_1 to symbolically execute e_2.

In order to enable semi-automated reasoning about C programs, we integrate our symbolic executor into a *verification condition generator (vcgen)*. Our vcgen does not merely turn programs into proof goals, but constructs the proof goals only as long as it can discharge goals automatically using our symbolic executor. When an attempt to use the symbolic executor fails, our vcgen will return a new goal, from which the vcgen can be called back again after the user helped out. This approach is useful when integrated into an interactive theorem prover.

We prove soundness of the symbolic executor and verification condition generator with respect to a refined version of the separation logic by Krebbers [29,30]. Our new logic has been developed on top of the Iris framework [26,24,33,25], and thereby inherits all advanced features of Iris (like its expressive support for ghost state and invariants), without having to model these explicitly. To make our new logic better suited for proving the correctness of the symbolic executor and verification condition generator, our new logic comes with a weakest precondition connective instead of Hoare triples as in Krebbers's original logic.

To streamline the soundness proof of our new program logic, we give a new *monadic definitional translation* of a subset of C relevant for non-determinism and sequence points into an ML-style functional language with concurrency. Contrary to the direct style operational semantics for a subset of C by Krebbers [29,30], our approach leads to a semantics that is both easier to understand, and easier to extend with additional language features.

We have mechanized our whole development in the Coq interactive theorem prover. The symbolic executor and verification condition generator are defined as computable functions in Coq, and have been integrated into tactics in the Iris Proof Mode/MoSeL framework [34,32]. To obtain end-to-end correctness, we mechanized the proofs of soundness of our symbolic executor and verification condition generator with respect to our new separation logic and new monadic definitional semantics for a subset of C. The Coq development is available at [18].

Contributions. We describe an approach to semi-automatically prove the absence of undefined behavior in a given C program for *any* evaluation order. While doing so, we make the following contributions:

- We define λMC: a small C-style language with a semantics by a monadic translation into an ML-style functional language with concurrency (§ 2);
- We present a separation logic with weakest preconditions for λMC based on the separation logic for non-determinism in C by Krebbers [29,30] (§ 3);
- We prove soundness of our separation logic with weakest preconditions by giving a modular model using the Iris framework [26,24,33,25] (§ 4);

– We present a new symbolic executor that not only computes the postcondition of a C expression, but also a *frame*, used to determine how resources should be distributed among subexpressions (§ 5);
– On top of our symbolic executor, we define a verification condition generator that enables semi-automated proofs using an interactive theorem prover (§6);
– We demonstrate that our approach can be implemented and proved sound using Coq for a superset of the λMC language considered in this paper (§ 7).

2 λMC: A Monadic Definitional Semantics of C

In this section we describe a small C-style language called λMC, which features non-determinism in expressions. We define its semantics by translation into a ML-style functional language with concurrency called HeapLang.

We briefly describe the λMC source language (§ 2.1) and the HeapLang target language (§ 2.2) of the translation. Then we describe the translation scheme itself (§ 2.3). We explain in several steps how to exploit concurrency and monadic programming to give a concise and clear definitional semantics.

2.1 The Source Language λMC

The syntax of our source language called λMC is as follows:

$$\begin{aligned} \mathtt{v} \in \mathsf{val} &::= \mathtt{z} \mid \mathtt{f} \mid \mathtt{l} \mid \mathtt{NULL} \mid (\mathtt{v}_1, \mathtt{v}_2) \mid () && (\mathtt{z} \in \mathbb{Z},\ \mathtt{l} \in \mathsf{Loc}) \\ \mathtt{e} \in \mathsf{expr} &::= \mathtt{v} \mid \mathtt{x} \mid (\mathtt{e}_1, \mathtt{e}_2) \mid \mathtt{e.1} \mid \mathtt{e.2} \mid \mathtt{e}_1 \circledcirc \mathtt{e}_2 \mid && (\circledcirc \in \{+, -, \dots\}) \\ & \quad \mathtt{x} \leftarrow \mathtt{e}_1 ; \mathtt{e}_2 \mid \mathtt{if(e_1)\{e_2\}\{e_3\}} \mid \mathtt{while(e_1)\{e_2\}} \mid \mathtt{e_1(e_2)} \mid \\ & \quad \mathtt{alloc(e)} \mid \mathtt{*e} \mid \mathtt{e}_1 = \mathtt{e}_2 \mid \mathtt{free(e)} \end{aligned}$$

The values include integers, `NULL` pointers, concrete locations `l`, function pointers `f`, structs with two fields (tuples), and the unit value `()` (for functions without return value). There is a global list of function definitions, where each definition is of the form `f(x){e}`. Most of the expression constructs resemble standard C notation, with some exceptions. We do not differentiate between expressions and statements to keep our language uniform. As such, if-then-else and sequencing constructs are not duplicated for both expressions and statements. Moreover, we do not differentiate between *lvalues* and *rvalues* [22, 6.3.2.1]. Hence, there is no address operator `&`, and, similarly to ML, the load (`*e`) and assignment ($\mathtt{e}_1 = \mathtt{e}_2$) operators take a reference as their first argument.

The *sequenced bind* operator $\mathtt{x} \leftarrow \mathtt{e}_1 ; \mathtt{e}_2$ generalizes the normal sequencing operator $\mathtt{e}_1 ; \mathtt{e}_2$ of C by binding the result of $\mathtt{e}_1$ to the variable `x` in $\mathtt{e}_2$. As such, $\mathtt{x} \leftarrow \mathtt{e}_1 ; \mathtt{e}_2$ can be thought of as the declaration of an immutable local variable `x`. We omit mutable local variables for now, but these can be easily added as an extension to our method, as shown in § 7. We write $\mathtt{e}_1 ; \mathtt{e}_2$ for a sequenced bind $\_ \leftarrow \mathtt{e}_1 ; \mathtt{e}_2$ in which we do not care about the return value of $\mathtt{e}_1$.

To focus on the key topics of the paper—non-determinism and the sequence point restriction—we take a minimalistic approach and omit most other features

of C. Notably, we omit non-local control (return, break, continue, and goto). Our memory model is simplified; it only supports structs with two fields (tuples), but no arrays, unions, or machine integers. In § 7 we show that some of these features (arrays, pointer arithmetic, and mutable local variables) can be incorporated.

2.2 The Target Language HeapLang

The target language of our definitional semantics of λMC is an ML-style functional language with concurrency primitives and a call-by-value semantics. This language, called HeapLang, is included as part of the Iris Coq development [21]. The syntax is as follows:

$$v \in \mathit{Val} ::= z \mid \texttt{true} \mid \texttt{false} \mid \texttt{rec}\ f\ x = e \mid \ell \mid () \mid \ldots \qquad (z \in \mathbb{Z}, \ell \in \mathit{Loc})$$
$$e \in \mathit{Expr} ::= v \mid x \mid e_1\ e_2 \mid \texttt{ref}(e) \mid !_{\mathsf{HL}}\, e \mid e_1 :=_{\mathsf{HL}} e_2 \mid \texttt{assert}(e) \mid$$
$$e_1 \mid\mid_{\mathsf{HL}} e_2 \mid \texttt{newmutex} \mid \texttt{acquire} \mid \texttt{release} \mid \ldots$$

The language contains some concurrency primitives that we will use to model non-determinism in λMC. Those primitives are $(||_{\mathsf{HL}})$, `newmutex`, `acquire`, and `release`. The first primitive is the parallel composition operator, which executes expressions e_1 and e_2 in parallel, and returns a tuple of their results. The expression `newmutex` () creates a new mutex. If *lk* is a mutex that was created this way, then `acquire` *lk* tries to acquire it and blocks until no other thread is using *lk*. An acquired mutex can be released using `release` *lk*.

2.3 The Monadic Definitional Semantics of λMC

We now give the semantics of λMC by translation into HeapLang. The translation is carried out in several stages, each iteration implementing and illustrating a specific aspect of C. First, we model non-determinism in expressions by concurrency, parallelizing execution of subexpressions (step 1). After that, we add checks for sequence point violations in the translation of the assignment and dereferencing operations (step 2). Finally, we add function calls and demonstrate how the translation can be simplified using a monadic notation (step 3).

Step 1: Non-determinism via Parallel Composition. We model the unspecified evaluation order in binary expressions like $\mathsf{e}_1 + \mathsf{e}_2$ and $\mathsf{e}_1 = \mathsf{e}_2$ by executing the subexpressions in parallel using the $(||_{\mathsf{HL}})$ operator:

$$[\![\mathsf{e}_1 + \mathsf{e}_2]\!] \triangleq \texttt{let}\ (v_1, v_2) = [\![\mathsf{e}_1]\!] \mid\mid_{\mathsf{HL}} [\![\mathsf{e}_2]\!]\ \texttt{in}\ v_1 +_{\mathsf{HL}} v_2$$

$$[\![\mathsf{e}_1 = \mathsf{e}_2]\!] \triangleq \texttt{let}\ (v_1, v_2) = [\![\mathsf{e}_1]\!] \mid\mid_{\mathsf{HL}} [\![\mathsf{e}_2]\!]\ \texttt{in}$$
$$\texttt{match}\ v_1\ \texttt{with}$$
$$\mid \texttt{None} \to \texttt{assert(false)} \quad \texttt{(* NULL pointer *)}$$
$$\mid \texttt{Some}\ l \to \texttt{match}\ !_{\mathsf{HL}}\, l\ \texttt{with}$$
$$\mid \texttt{None} \to \texttt{assert(false)} \quad \texttt{(* Use after free *)}$$
$$\mid \texttt{Some}\ \_ \to l :=_{\mathsf{HL}} \texttt{Some}\ v_2;\ v_2$$

Since our memory model is simple, the value interpretation is straightforward:

$$[\![\mathtt{z}]\!]_{val} \triangleq \mathtt{z} \quad (\text{if } \mathtt{z} \in \mathbb{Z}) \qquad [\![\mathtt{NULL}]\!]_{val} \triangleq \mathtt{None}$$
$$[\![(\mathtt{v}_1, \mathtt{v}_2)]\!]_{val} \triangleq ([\![\mathtt{v}_1]\!]_{val}, [\![\mathtt{v}_2]\!]_{val}) \qquad [\![()]\!]_{val} \triangleq () \qquad [\![\mathtt{l}]\!]_{val} \triangleq \mathtt{Some\ l}$$

The only interesting case is the translation of locations. Since there is no concept of a `NULL` pointer in HeapLang, we use the option type to distinguish `NULL` pointers from concrete locations (`l`). The interpretation of assignments thus contains a pattern match to check that no `NULL` pointers are dereferenced. A similar check is performed in the interpretation of the load operation (`*e`). Moreover, each location contains an option to distinguish freed from active locations.

Step 2: Sequence Points. So far we have not accounted for undefined behavior due to sequence point violations. For instance, the program `(x = 3) + (x = 4)` gets translated into a HeapLang expression that updates the value of the location `x` non-deterministically to either 3 or 4, and returns 7. However, in C, the behavior of this program is *undefined*, as it exhibits a sequence point violation: there is a write conflict for the location `x`.

To give a semantics for sequence point violations, we follow the approach by Norrish [44], Ellison and Rosu [17], and Krebbers [29,30]. We keep track of a set of locations that have been written to since the last sequence point. We refer to this set as the *environment* of our translation, and represent it using a global variable *env* of the type `mset` *Loc*. Because our target language HeapLang is concurrent, all updates to the environment *env* must be executed *atomically*, *i.e.,* inside a critical section, which we enforce by employing a global mutex *lk*. The interpretation of assignments $\mathtt{e}_1 = \mathtt{e}_2$ now becomes:

```
⟦e1 = e2⟧ ≜ let (v1, v2) = ⟦e1⟧ ||HL ⟦e2⟧ in
            acquire lk;
            match v1 with
             | None → assert(false)     (* NULL pointer *)
             | Some l →
                assert(¬mset_member l env); (* Seq. point violation *)
                match !HL l with
                 | None → assert(false)  (* Use after free *)
                 | Some _ → mset_add l env; l :=HL Some v2;
            release lk; v2
```

Whenever we assign to (or read from) a location l, we check if the location l is not already present in the environment *env*. If the location l is present, then it was already written to since the last sequence point. Hence, accessing the location constitutes undefined behavior (see the `assert` in the interpretation of assignments above). In the interpretation of assignments, we furthermore insert the location l into the environment *env*.

$$\begin{aligned}
\texttt{ret}\, e &\triangleq \lambda _\ _.\ e \\
e_1 \mid\mid e_2 &\triangleq \lambda\, env\ lk.\ (e_1\ env\ lk) \mid\mid_{\mathsf{HL}} (e_2\ env\ lk) \\
x \leftarrow e_1;\ e_2 &\triangleq \lambda\, env\ lk.\ \texttt{let}\ x = e_1\ env\ lk\ \texttt{in}\ e_2\ env\ lk \\
\texttt{atomic_env}\, e &\triangleq \lambda\, env\ lk.\ \texttt{acquire}\ lk; \texttt{let}\ a = e\ env\ \texttt{in}\ \texttt{release}\ lk; a \\
\texttt{atomic}\, e &\triangleq \lambda\, env\ lk.\ \texttt{acquire}\ lk; \texttt{let}\ a = e\ env\ (\texttt{newmutex}\ ())\ \texttt{in}\ \texttt{release}\ lk; a \\
\texttt{run}(e) &\triangleq e\ (\texttt{mset_create}\ ())\ (\texttt{newmutex}\ ())
\end{aligned}$$

Fig. 1. The monadic combinators.

In order to make sure that one can access a variable again after a sequence point, we define the *sequenced bind* operator $\texttt{x} \leftarrow \texttt{e}_1\,;\texttt{e}_2$ as follows:

$$[\![\texttt{x} \leftarrow \texttt{e}_1\,;\texttt{e}_2]\!] \triangleq \texttt{let}\ x = [\![\texttt{e}_1]\!]\ \texttt{in}\ \texttt{acquire}\ lk;\ \texttt{mset\_clear}\ env;\ \texttt{release}\ lk;\ [\![\texttt{e}_2]\!]$$

After we finished executing the expression $\texttt{e}_1$, we clear the environment *env*, so that all locations are accessible in $\texttt{e}_2$ again.

Step 3: Non-Interleaved Function Calls. As the final step, we present the correct translation scheme for function calls. Unlike the other expressions, function calls are not interleaved during the execution of subexpressions [22, 6.5.2.2p10]. For instance, in the program `f() + g()` the possible orders of execution are: either all the instructions in `f()` followed by all the instructions in `g()`, or all the instructions in `g()` followed by all the instructions in `f()`.

To model this, we execute each function call *atomically*. In the previous step we used a global mutex for guarding the access to the environment. We could use that mutex for function calls too. However, reusing a single mutex for entering each critical section would not work because a body of a function may contain invocations of other functions. To that extent, we use multiple mutexes to reflect the hierarchical structure of function calls.

To handle multiple mutexes, each C expression is interpreted as a HeapLang function that receives a mutex and returns its result. That is, each C expression is modeled by a monadic expression in the *reader monad* $M(A) \triangleq \texttt{mset}\, Loc \to \texttt{mutex} \to A$. For consistency's sake, we now also use the monad to thread through the reference to the environment ($\texttt{mset}\, Loc$), instead of using a global variable *env* as we did in the previous step.

We use a small set of monadic combinators, shown in Figure 1, to build the translation in a more abstract way. The return and bind operators are standard for the reader monad. The parallel operator runs two monadic expressions concurrently, propagating the environment and the mutex. The `atomic` combinator invokes a monadic expression with a fresh mutex. The `atomic_env` combinator atomically executes its body with the current environment as an argument. The `run` function executes the monadic computation by instantiating it with

$$
\begin{aligned}
[\![\mathtt{e}_1 + \mathtt{e}_2]\!] &\triangleq (v_1, v_2) \leftarrow [\![\mathtt{e}_1]\!] \mid\mid [\![\mathtt{e}_2]\!];\ \mathtt{ret}\,(v_1 +_{\mathsf{HL}} v_2)\\
[\![\mathtt{e}_1 = \mathtt{e}_2]\!] &\triangleq (v_1, v_2) \leftarrow [\![\mathtt{e}_1]\!] \mid\mid [\![\mathtt{e}_2]\!];\\
&\quad \mathtt{atomic\_env}\,(\lambda\ env.\\
&\quad\ \ \mathtt{match}\ v_1\ \mathtt{with}\\
&\quad\ \ \mid \mathtt{None} \rightarrow \mathtt{assert(false)} \quad \mathtt{(*\ NULL\ pointer\ *)}\\
&\quad\ \ \mid \mathtt{Some}\ l \rightarrow\\
&\quad\ \ \ \ \mathtt{assert}(\neg\mathtt{mset\_member}\ l\ env);\ \mathtt{(*\ Seq.\ point\ violation\ *)}\\
&\quad\ \ \ \ \mathtt{match}\ !_{\mathsf{HL}}\, l\ \mathtt{with}\\
&\quad\ \ \ \ \mid \mathtt{None} \rightarrow \mathtt{assert(false)} \quad \mathtt{(*\ Use\ after\ free\ *)}\\
&\quad\ \ \ \ \mid \mathtt{Some}\ \_ \rightarrow \mathtt{mset\_add}\ l\ env;\ l :=_{\mathsf{HL}} \mathtt{Some}\ v_2;\ \mathtt{ret}\ v_2)\\
[\![\mathtt{x} \leftarrow \mathtt{e}_1\,;\mathtt{e}_2]\!] &\triangleq x \leftarrow [\![\mathtt{e}_1]\!];\ \_ \leftarrow (\mathtt{atomic\_env\ mset\_clear});\ [\![\mathtt{e}_2]\!]\\
[\![\mathtt{e}_1\mathtt{(e}_2\mathtt{)}]\!] &\triangleq (f, a) \leftarrow [\![\mathtt{e}_1]\!] \mid\mid [\![\mathtt{e}_2]\!];\ \mathtt{atomic}\,(\mathtt{atomic\_env\ mset\_clear}; f\ a)\\
[\![\mathtt{f(x)\{e\}}]\!] &\triangleq \mathtt{let\ rec\ f\ x} = v \leftarrow [\![\mathtt{e}]\!];\ \_ \leftarrow (\mathtt{atomic\_env\ mset\_clear});\ \mathtt{ret}\ v
\end{aligned}
$$

Fig. 2. Selected clauses from the monadic definitional semantics.

a fresh mutex and a new environment. Selected clauses for the translation are presented in Figure 2. The translation of the binary operations remains virtually unchanged, except for the usage of monadic parallel composition instead of the standard one. The translation for the assignment and the sequenced bind uses the `atomic_env` combinator for querying and updating the environment. We also have to adapt our translation of values, by wrapping it in $\mathtt{ret}: [\![\mathtt{v}]\!] \triangleq \mathtt{ret}\ [\![\mathtt{v}]\!]_{val}$.

A global function definition `f(x){e}` is translated as a top level let-binding. A function call is then just an atomically executed function invocation in HeapLang, modulo the fact that the function pointer and the arguments are computed in parallel. In addition, sequence points occur at the beginning of each function call and at the end of each function body [22, Annex C], and we reflect that in our translation by clearing the environment at appropriate places.

Our semantics by translation can easily be extended to cover other features of C, *e.g.,* a more advanced memory model (see § 7). However the fragment presented here already illustrates the challenges that non-determinism and sequence point violations pose for verification. In the next section we describe a logic for reasoning about the semantics by translation given in this section.

3 Separation Logic with Weakest Preconditions for λMC

In this section we present a separation logic with weakest precondition propositions for reasoning about λMC programs. The logic tackles the main features of our semantics—non-determinism in expressions evaluation and sequence point violations. We will discuss the high-level rules of the logic pertaining to C connectives by going through a series of small examples.

The logic presented here is similar to the separation logic by Krebbers [29], but it is given in a weakest precondition style, and moreover, it is constructed *synthetically* on top of the separation logic framework Iris [26,24,33,25], whereas the logic by Krebbers [29] is interpreted directly in a bespoke model.

The following grammar defines the formulas of the logic:

$$P, Q \in \mathsf{Prop} ::= \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \mathtt{v}_1 = \mathtt{v}_2 \mid \mathtt{l} \overset{q}{\mapsto}_{\xi} v \mid \qquad (q \in (0,1])$$
$$P * Q \mid P \mathrel{-\!\!*} Q \mid \mathsf{wp}\, e\, \{\Phi\} \mid \ldots \qquad (\xi \in \{L, U\})$$

Most of the connectives are commonplace in separation logic, with the exception of the modified points-to connective, which we describe in this section.

As is common, Hoare triples $\{P\}\, \mathsf{e}\, \{\Phi\}$ are syntactic sugar for $P \vdash \mathsf{wp}\, \mathsf{e}\, \{\Phi\}$. The weakest precondition connective $\mathsf{wp}\, \mathsf{e}\, \{\Phi\}$ states that the program e is safe (the program has defined behavior), and if e terminates to a value $\mathtt{v}$, then $\mathtt{v}$ satisfies the predicate Φ. We write $\mathsf{wp}\, \mathsf{e}\, \{\mathtt{v}.\, \Phi\, \mathtt{v}\}$ for $\mathsf{wp}\, \mathsf{e}\, \{\lambda\, \mathtt{v}.\, \Phi\, \mathtt{v}\}$.

Contrary to the paper by Krebbers [29], we use weakest preconditions instead of Hoare triples throughout this paper. There are several reasons for doing so:

1. We do not have to manipulate the preconditions explicitly, *e.g.,* by applying the consequence rule to the precondition.
2. The soundness of our symbolic executor (Theorem 5.1) can be stated more concisely using weakest precondition propositions.
3. It is more convenient to integrate weakest preconditions into the Iris Proof Mode/MoSeL framework in Coq that we use for our implementation (§ 7).

A selection of rules is presented in Figure 3. Each inference rule $\dfrac{P_1 \ \ldots \ P_n}{Q}$ in this paper should be read as the entailment $P_1 * \ldots * P_n \vdash Q$. We now explain and motivate the rules of our logic.

Non-Determinism. In the introduction (§ 1) we have already shown the rule for addition from Krebbers's logic [29], which was written using Hoare triples. Using weakest preconditions, the corresponding rule (WP-BIN-OP) is:

$$\frac{\mathsf{wp}\, \mathsf{e}_1\, \{\Psi_1\} \qquad \mathsf{wp}\, \mathsf{e}_2\, \{\Psi_2\} \qquad (\forall \mathtt{w}_1 \mathtt{w}_2.\, \Psi_1\, \mathtt{w}_1 * \Psi_2\, \mathtt{w}_2 \mathrel{-\!\!*} \Phi(\mathtt{w}_1\, [\![\circledcirc]\!]\, \mathtt{w}_2))}{\mathsf{wp}\, (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\, \{\Phi\}}$$

This rule closely resembles the usual rule for parallel composition in ordinary concurrent separation logic [46]. This should not be surprising, as we have given a definitional semantics to binary operators using the parallel composition operator. It is important to note that the premises WP-BIN-OP are combined using the *separating conjunction* $*$. This ensures that the weakest preconditions $\mathsf{wp}\, \mathsf{e}_1\, \{\Psi_1\}$ and $\mathsf{wp}\, \mathsf{e}_2\, \{\Psi_2\}$ for the subexpressions e_1 and e_2 are verified with respect to disjoint resources. As such they do not interfere with each other, and can be evaluated in parallel without causing sequence point violations.

To see how one can use the rule WP-BIN-OP, let us verify $P \vdash \mathsf{wp}\, (\mathsf{e}_1 + \mathsf{e}_2)\, \{\Phi\}$. That is, we want to show that $(\mathsf{e}_1 + \mathsf{e}_2)$ satisfies the postcondition Φ assuming

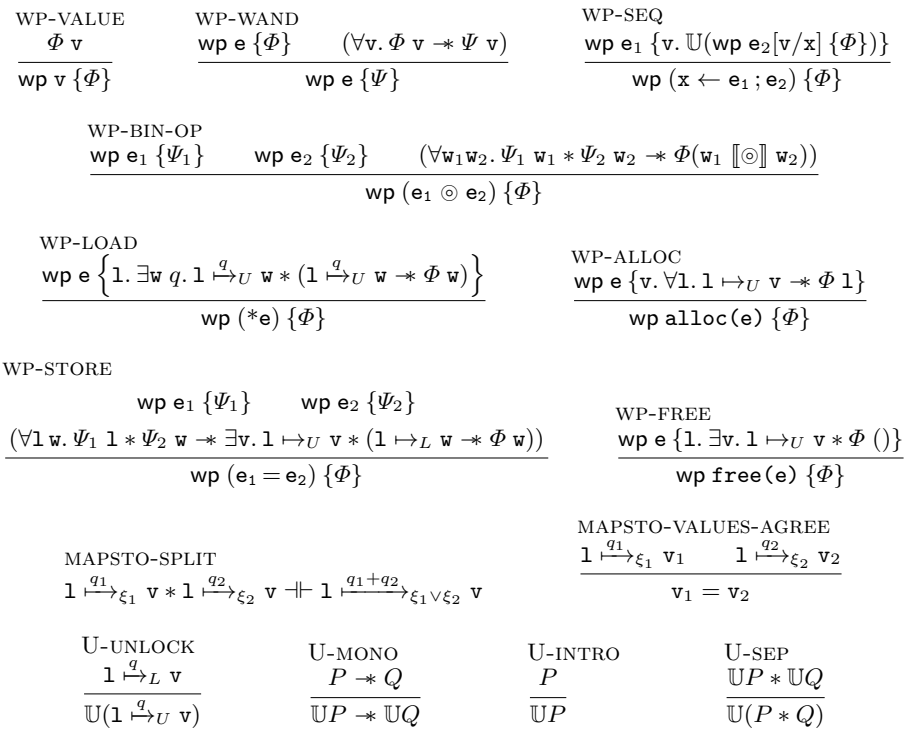

Fig. 3. Selected rules for weakest preconditions.

the precondition P. This goal can be proven by separating the precondition P into disjoint parts $P_1 * P_2 * R \dashv\vdash P$. Then using WP-BIN-OP the goal can be reduced to proving $P_i \vdash \mathsf{wp}\ \mathtt{e}_i\ \{\Psi_i\}$ for $i \in \{0, 1\}$, and $R * \Psi_1\ \mathtt{w}_1 * \Psi_2\ \mathtt{w}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)$ for any return values $\mathtt{w}_i$ of the expressions $\mathtt{e}_i$.

Fractional Permissions. Separation logic includes the *points-to connective* $\mathtt{l} \mapsto \mathtt{v}$, which asserts unique ownership of a location $\mathtt{l}$ with value $\mathtt{v}$. This connective is used to specify the behavior of stateful operations, which becomes apparent in the following proposed rule for load:

$$\frac{\mathsf{wp}\ \mathtt{e}\ \{\mathtt{l}.\ \exists \mathtt{w}.\ \mathtt{l} \mapsto \mathtt{w} * (\mathtt{l} \mapsto \mathtt{w} \mathrel{-\!\!*} \Phi\ \mathtt{w})\}}{\mathsf{wp}\ (*\mathtt{e})\ \{\Phi\}}$$

In order to verify $*\mathtt{e}$ we first make sure that $\mathtt{e}$ evaluates to a location $\mathtt{l}$, and then we need to provide the points-to connective $\mathtt{l} \mapsto \mathtt{w}$ for some value stored at the location. This rule, together with WP-VALUE, allows for verification of simple programs like $\mathtt{l} \mapsto \mathtt{v} \vdash \mathsf{wp}\ (*\mathtt{l})\ \{\mathtt{w}.\ \mathtt{w} = \mathtt{v} * \mathtt{l} \mapsto \mathtt{v}\}$.

However, the rule above is too weak. Suppose that we wish to verify the program $*\mathtt{l} + *\mathtt{l}$ from the precondition $\mathtt{l} \mapsto \mathtt{v}$. According to WP-BIN-OP, we have

to separate the proposition $\mathtt{l} \mapsto \mathtt{v}$ into two disjoint parts, each used to verify the load operation. In order to enable sharing of points-to connectives we use *fractional permissions* [8,7]. In separation logic with fractional permissions each points-to connective is annotated with a fraction $q \in (0, 1]$, and the resources can be split in accordance with those fractions:

$$\mathtt{l} \overset{q_1+q_2}{\longmapsto} \mathtt{v} \dashv\vdash \mathtt{l} \overset{q_1}{\longmapsto} \mathtt{v} * \mathtt{l} \overset{q_2}{\longmapsto} \mathtt{v}.$$

A connective $\mathtt{l} \overset{1}{\mapsto} \mathtt{v}$ provides a unique ownership of the location, and we refer to it as a *write permission*. A points-to connective with $q \leq 1$ provides shared ownership of the location, referred to as a *read permission*. By convention, we write $\mathtt{l} \mapsto \mathtt{v}$ to denote the write permission $\mathtt{l} \overset{1}{\mapsto} \mathtt{v}$.

With fractional permissions at hand, we can relax the proposed load rule, by allowing to dereference a location even if we only have a read permission:

$$\frac{\mathsf{wp}\ \mathsf{e}\ \left\{\mathtt{l}.\ \exists \mathtt{w}\, q.\, \mathtt{l} \overset{q}{\mapsto} \mathtt{w} * (\mathtt{l} \overset{q}{\mapsto} \mathtt{w} \mathrel{-\!\!*} \Phi\ \mathtt{w})\right\}}{\mathsf{wp}\ (*\mathsf{e})\ \{\Phi\}}$$

This corresponds to the intuition that multiple subexpressions can safely dereference the same location, but not write to them.

Using the rule above we can verify $\mathtt{l} \mapsto 1 \vdash \mathsf{wp}\ (*\mathtt{l} + *\mathtt{l})\ \{\mathtt{v}.\, \mathtt{v} = 2 * \mathtt{l} \mapsto 1\}$ by splitting the assumption into $\mathtt{l} \overset{0.5}{\longmapsto} 1 * \mathtt{l} \overset{0.5}{\longmapsto} 1$ and first applying WP-BIN-OP with Ψ_1 and Ψ_2 being $\lambda\, \mathtt{v}.\, (\mathtt{v} = 1) * \mathtt{l} \overset{0.5}{\longmapsto} 1$. Then we apply WP-LOAD on both subgoals. After that, we can use MAPSTO-SPLIT to prove the remaining formula:

$$(\mathtt{v}_1 = 1) * \mathtt{l} \overset{0.5}{\longmapsto} 1 * (\mathtt{v}_2 = 1) * \mathtt{l} \overset{0.5}{\longmapsto} 1 \ \vdash\ (\mathtt{v}_1 + \mathtt{v}_2 = 2) * \mathtt{l} \mapsto 1.$$

The Assignment Operator. The second main operation that accesses the heap is the assignment operator $\mathsf{e}_1 = \mathsf{e}_2$. The arguments on the both sides of the assignment are evaluated in parallel, and a points-to connective is required to perform an update to the heap. A naive version of the assignment rule can be obtained by combining the binary operation rule and the load rule:

$$\frac{\mathsf{wp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{wp}\ \mathsf{e}_2\ \{\Psi_2\} \qquad (\forall \mathtt{l}\, \mathtt{w}.\, \Psi_1\ \mathtt{l} * \Psi_2\ \mathtt{w} \mathrel{-\!\!*} \exists \mathtt{v}.\, \mathtt{l} \mapsto \mathtt{v} * (\mathtt{l} \mapsto \mathtt{w} \mathrel{-\!\!*} \Phi\ \mathtt{w}))}{\mathsf{wp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\Phi\}}$$

The write permission $\mathtt{l} \mapsto \mathtt{v}$ can be obtained by combining the resources of both sides of the assignment. This allows us to verify programs like $\mathtt{l} = *\mathtt{l} + *\mathtt{l}$.

However, the rule above is unsound, because it fails to account for sequence point violations. We could use the rule above to prove safety of undefined programs, *e.g.,* the program $\mathtt{l} = (\mathtt{l} = 3)$.

To account for sequence point violations we decorate the points-to connectives $\mathtt{l} \overset{q}{\mapsto}_{\xi} \mathtt{v}$ with *access levels* $\xi \in \{L, U\}$. These have the following semantics: we can read from and write to a location that is unlocked (U), and the location becomes locked (L) once someone writes to it. Proposition $\mathtt{l} \overset{q}{\mapsto}_{U} \mathtt{v}$

(resp. $\mathtt{l} \overset{q}{\mapsto}_L \mathtt{v}$) asserts ownership of the unlocked (resp. locked) location $\mathtt{l}$. We refer to such propositions as *lockable points-to connectives*. Using lockable points-to connectives we can formulate the correct assignment rule:

$$\frac{\mathsf{wp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{wp}\ \mathsf{e}_2\ \{\Psi_2\} \qquad (\forall \mathtt{l}\ \mathtt{w}.\ \Psi_1\ \mathtt{l} * \Psi_2\ \mathtt{w} \mathrel{-\!\!*} \exists \mathtt{v}.\ \mathtt{l} \mapsto \mathtt{v} * (\mathtt{l} \mapsto_L \mathtt{w} \mathrel{-\!\!*} \Phi\ \mathtt{w}))}{\mathsf{wp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\Phi\}}$$

The set $\{L, U\}$ has a lattice structure with $L \leq U$, and the levels can be combined with a join operation, see MAPSTO-SPLIT. By convention, $\mathtt{l} \overset{q}{\mapsto} \mathtt{v}$ denotes $\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}$.

The Unlocking Modality. As locations become locked after using the assignment rule, we wish to unlock them in order to perform further heap operations. For instance, in the expression $\mathtt{l} = \mathtt{4}\,;\mathtt{*l}$ the location $\mathtt{l}$ becomes unlocked after the sequence point ";" between the store and the dereferencing operations. To reflect this in the logic, we use the rule WP-SEQ which features the *unlocking modality* $\mathbb{U}$ (which is called the unlocking assertion in [29, Definition 5.6]):

$$\frac{\mathsf{wp}\ \mathsf{e}_1\ \{\_.\ \mathbb{U}(\mathsf{wp}\ \mathsf{e}_2\ \{\Phi\})\}}{\mathsf{wp}\ (\mathsf{e}_1\,;\mathsf{e}_2)\ \{\Phi\}}$$

Intuitively, $\mathbb{U}P$ states that P holds, after unlocking all locations. The rules of $\mathbb{U}$ in Figure 3 allow one to turn $(P_1 * \ldots * P_m) * (\mathtt{l}_1 \mapsto_L \mathtt{v}_1 * \ldots * \mathtt{l}_m \mapsto_L \mathtt{v}_m) \vdash \mathbb{U}Q$ into $(P_1 * \ldots * P_m) * (\mathtt{l}_1 \mapsto_U \mathtt{v}_1 * \ldots * \mathtt{l}_m \mapsto_U \mathtt{v}_m) \vdash Q$. This is done by applying either U-UNLOCK or U-INTRO to each premise; then collecting all premises into one formula under $\mathbb{U}$ by U-SEP; and finally, applying U-MONO to the whole sequent.

4 Soundness of Weakest Preconditions for λMC

In this section we prove adequacy of the separation logic with weakest preconditions for λMC as presented in § 3. We do this by giving a model using the Iris framework that is structured in a similar way as the translation that we gave in § 2. This translation consisted of three layers: the target HeapLang language, the monadic combinators, and the λMC operations themselves. In the model, each corresponding layer abstracts from the details of the previous layer, in such a way that we never have to break the abstraction of a layer. At the end, putting all of this together, we get the following adequacy statement:

Theorem 4.1 (Adequacy of Weakest Preconditions). *If* $\mathsf{wp}\ \mathsf{e}\ \{\Phi\}$ *is derivable, then* $\mathtt{e}$ *has no undefined behavior for any evaluation order. In other words,* $\mathtt{run}(\mathtt{e})$ *does not assert false.*

The proof of the adequacy theorem closely follows the layered structure, by combining the correctness of the monadic `run` combinator with adequacy of HeapLang in Iris [25, Theorem 6]. The rest of this section is organized as:

1. Because our translation targets HeapLang, we start by recalling the separation logic with weakest preconditions, for HeapLang part of Iris (§ 4.1).

$$(\ell \mapsto_{\mathsf{HL}} v) * (\ell \mapsto_{\mathsf{HL}} v \mathrel{-\!\!*} \Phi\ v) \vdash \mathsf{wp}_{\mathsf{HL}}\ !_{\mathsf{HL}}\ \ell\ \{\Phi\}$$
$$(\ell \mapsto_{\mathsf{HL}} v) * (\ell \mapsto_{\mathsf{HL}} w \mathrel{-\!\!*} \Phi\ ()) \vdash \mathsf{wp}_{\mathsf{HL}}\ \ell :=_{\mathsf{HL}} w\ \{\Phi\}$$

$$\text{WP}_{\mathsf{HL}}\text{-BIND}\quad \frac{\mathsf{wp}_{\mathsf{HL}}\ e\ \{v.\ \mathsf{wp}_{\mathsf{HL}}\ K[v]\ \{\Phi\}\}}{\mathsf{wp}_{\mathsf{HL}}\ K[e]\ \{\Phi\}}$$

$$R * (\forall \gamma\ lk.\ \mathsf{is\_mutex}(\gamma, lk, R) \mathrel{-\!\!*} \Phi\ lk) \vdash \mathsf{wp}_{\mathsf{HL}}\ \texttt{newmutex}\ ()\ \{\Phi\}$$
$$\mathsf{is\_mutex}(\gamma, lk, R) * (R * \mathsf{locked}(\gamma) \mathrel{-\!\!*} \Phi\ ()) \vdash \mathsf{wp}_{\mathsf{HL}}\ \texttt{acquire}\ lk\ \{\Phi\}$$
$$\mathsf{is\_mutex}(\gamma, lk, R) * R * \mathsf{locked}(\gamma) * \Phi\ () \vdash \mathsf{wp}_{\mathsf{HL}}\ \texttt{release}\ lk\ \{\Phi\}$$
$$\mathsf{is\_mutex}(\gamma, lk, R) * \mathsf{is\_mutex}(\gamma, lk, R) \dashv\vdash \mathsf{is\_mutex}(\gamma, lk, R) \qquad \text{(ISMUTEX-DUPL)}$$

Fig. 4. Selected $\mathsf{wp}_{\mathsf{HL}}$ rules.

2. On top of the logic for HeapLang, we define a notion of weakest preconditions $\mathsf{wp}_{\mathsf{mon}}\ e\ \{\Phi\}$ for expressions e built from our monadic combinators (§ 4.2).
3. Next, we define the lockable points-to connective $\ell \overset{q}{\mapsto}_{\xi} v$ using Iris's machinery for custom ghost state (§ 4.3).
4. Finally, we define weakest preconditions for λMC by combining the weakest preconditions for monadic expressions with our translation scheme (§ 4.4).

4.1 Weakest Preconditions for HeapLang

We recall the most essential Iris connectives for reasoning about HeapLang programs: $\mathsf{wp}_{\mathsf{HL}}\ e\ \{\Phi\}$ and $\ell \mapsto_{\mathsf{HL}} v$, which are the HeapLang weakest precondition proposition and the HeapLang points-to connective, respectively. Other Iris connectives are described in [6, Section 8.1] or [25,33]. An example rule is the store rule for HeapLang, shown in Figure 4. The rule requires a points-to connective $\ell \mapsto_{\mathsf{HL}} v$, and the user receives the updated points-to connective $\ell \mapsto_{\mathsf{HL}} w$ back for proving $\Phi\ ()$. Note that the rule is formulated for a concrete location ℓ and a value w, instead of arbitrary expressions. This does not limit the expressive power; since the evaluation order in HeapLang is deterministic[3], arbitrary expressions can be handled using the WP$_{\mathsf{HL}}$-BIND rule. Using this rule, one can bind an expression e in an arbitrary evaluation context K. We can thus use the WP$_{\mathsf{HL}}$-BIND rule twice to derive a more general store rule for HeapLang:

$$\frac{\mathsf{wp}_{\mathsf{HL}}\ e_2\ \{w.\ \mathsf{wp}_{\mathsf{HL}}\ e_1\ \{\ell.\ (\exists v.\ \ell \mapsto_{\mathsf{HL}} v) * (\ell \mapsto_{\mathsf{HL}} w \mathrel{-\!\!*} \Phi\ ())\}\}}{\mathsf{wp}_{\mathsf{HL}}\ (e_1 :=_{\mathsf{HL}} e_2)\ \{\Phi\}}$$

To verify the monadic combinators and the translation of λMC operations in the upcoming sections § 4.2 and 4.4, we need the specifications for all the functions that we use, including those on mutable sets and mutexes. The rules for mutable sets are standard, and thus omitted. They involve the usual abstract predicate $\mathsf{is\_mset}(s, X)$ stating that the reference s represents a set with contents X. The rules for mutexes are presented in Figure 4. When a new mutex is created,

[3] And right-to-left, although our monadic translation does not rely on that.

$$\text{WP-RET}\quad \frac{\mathsf{wp}_{\mathsf{HL}}\, e\, \{\Phi\}}{\mathsf{wp}_{\mathsf{mon}}\, (\mathtt{ret}\, e)\, \{\Phi\}} \qquad \text{WP-BIND}\quad \frac{\mathsf{wp}_{\mathsf{mon}}\, e_1\, \{v.\, \mathsf{wp}_{\mathsf{mon}}\, e_2[v/x]\, \{\Phi\}\}}{\mathsf{wp}_{\mathsf{mon}}\, (x \leftarrow e_1;\, e_2)\, \{\Phi\}}$$

$$\text{WP-PAR}\quad \frac{\mathsf{wp}_{\mathsf{mon}}\, e_1\, \{\Psi_1\} \qquad \mathsf{wp}_{\mathsf{mon}}\, e_2\, \{\Psi_2\} \qquad (\forall w_1 w_2.\, \Psi_1\, w_1 * \Psi_2\, w_2 \mathrel{-\!\!*} \Phi\, (w_1, w_2))}{\mathsf{wp}_{\mathsf{mon}}\, (e_1 \mid\mid e_2)\, \{\Phi\}}$$

$$\text{WP-ATOMIC-ENV}\quad \frac{\forall env.\, \mathsf{env\_inv}(env) \mathrel{-\!\!*} \mathsf{wp}_{\mathsf{HL}}\, (v\, env)\, \{w.\, \mathsf{env\_inv}(env) * \Phi\, w\}}{\mathsf{wp}_{\mathsf{mon}}\, (\mathtt{atomic\_env}\, v)\, \{\Phi\}}$$

Fig. 5. Selected monadic $\mathsf{wp}_{\mathsf{mon}}$ rules.

a user gets access to a proposition $\mathsf{is\_mutex}(\gamma, lk, R)$, which states that the value lk is a mutex containing the resources R. This proposition can be duplicated freely (ISMUTEX-DUPL). A thread can acquire the mutex and receive the resources contained in it. In addition, the thread receives a token $\mathsf{locked}(\gamma)$ meaning that it has entered the critical section. When a thread leaves the critical section and releases the mutex, it has to give up both the token and the resources R.

4.2 Weakest Preconditions for Monadic Expressions

As a next step, we define a weakest precondition proposition $\mathsf{wp}_{\mathsf{mon}}\, e\, \{\Phi\}$ for a monadic expression e. The definition is constructed in the ambient logic, and it encapsulates the monadic operations in a separate layer. Due to that, we are able to carry out proofs of high-level specifications without breaking the abstraction (§ 4.4). The specifications for selected monadic operations in terms of $\mathsf{wp}_{\mathsf{mon}}$ are presented in Figure 5. We define the weakest precondition for a monadic expression e as follows:

$$\mathsf{wp}_{\mathsf{mon}}\, e\, \{\Phi\} \triangleq \mathsf{wp}_{\mathsf{HL}}\, e \left\{ g.\, \forall \gamma\ env\ lk.\, \mathsf{is\_mutex}(\gamma, lk, \mathsf{env\_inv}(env)) \mathrel{-\!\!*} \mathsf{wp}_{\mathsf{HL}}\, (g\ env\ lk)\, \{\Phi\} \right\}$$

The idea is that we first reduce e to a monadic value g. To perform this reduction we have the outermost $\mathsf{wp}_{\mathsf{HL}}$ connective in the definition of $\mathsf{wp}_{\mathsf{mon}}$. This monadic value is then evaluated with an arbitrary environment and an arbitrary mutex. Note that we universally quantify over any mutex lk to support nested locking in `atomic`. This definition is parameterized by an *environment invariant* $\mathsf{env\_inv}(env)$, which describes the resources accessible in the critical sections. We show how to define $\mathsf{env\_inv}$ in the next subsection.

Using this definition we derive the monadic rules in Figure 5. In a monad, the expression evaluation order is made explicit via the bind operation $x \leftarrow e_1; e_2$. To that extent, contrary to HeapLang, we no longer have a rule like WP_{HL}-BIND, which allows to bind an expression in a general evaluation context. Instead, we have the rule WP-BIND, which reflects that the only evaluation context we have is the monadic bind $x \leftarrow [\bullet]; e$.

$$\textsc{heap-alloc}\quad \frac{\ell \mapsto_{\mathsf{HL}} v \qquad \mathsf{full\_heap}(\sigma)}{\Rrightarrow \ell \mapsto_{U} v * \mathsf{full\_heap}(\sigma\,[\ell \leftarrow (U, v)])}$$

$$\textsc{heap-upd}\quad \frac{\ell \mapsto_{U} v \qquad \mathsf{full\_heap}(\sigma)}{\Rrightarrow \sigma(\ell) = (U, v) * \ell \mapsto_{\mathsf{HL}} v * (\forall v'\, \xi'.\, \ell \mapsto_{\mathsf{HL}} v' \mathrel{-\!\!*\!\!\!\Rrightarrow} \ell \mapsto_{\xi'} v' * \mathsf{full\_heap}(\sigma\,[\ell \leftarrow (\xi', v')]))}$$

Fig. 6. Selected rules of the lockable heap construction.

4.3 Modeling the Heap

The monadic rules in Figure 5 are expressive enough to derive some of the λMC-level rules, but we are still missing one crucial part: handling of the heap. In order to do that, we need to define lockable points-to connectives $\mathtt{l} \overset{q}{\mapsto}_{\xi} \mathtt{v}$ in such a way that they are linked to the HeapLang points-to connectives $\ell \mapsto_{\mathsf{HL}} v$.

The key idea is the following. The environment invariant env_inv of monadic weakest preconditions will track *all* HeapLang points-to connectives $\ell \mapsto_{\mathsf{HL}} v$ that have ever been allocated at the λMC level. Via Iris ghost state, we then connect this knowledge to the lockable points-to connectives $\mathtt{l} \overset{q}{\mapsto}_{\xi} \mathtt{v}$. We refer to the construction that allows us to carry this out as the *lockable heap*. Note that the description of lockable heap is fairly technical and requires an understanding of the ghost state mechanism in Iris.

A lockable heap is a map $\sigma : \mathit{Loc} \xrightarrow{\text{fin}} \{L, U\} \times \mathit{Val}$ that keeps track of the access levels and values associated with the locations. The connective $\mathsf{full\_heap}(\sigma)$ asserts the ownership of all the locations present in the domain of σ. Specifically, it asserts $\ell \mapsto_{\mathsf{HL}} v$ for each $\{\ell \leftarrow (\xi, v)\} \in \sigma$. The connective $\ell \overset{q}{\mapsto}_{\xi} v$ then states that $\{\ell \leftarrow (\xi, v)\}$ is part of the global lockable heap, and it asserts this with the fractional permission q. We treat the lockable heap as an opaque abstraction, whose exact implementation via Iris ghost state is described in the Coq formalization [18]. The main interface for the locking heap are the rules in Figure 6. The rule HEAP-ALLOC states that we can turn a HeapLang points-to connective $\ell \mapsto_{\mathsf{HL}} v$ into $\ell \mapsto_{\xi} v$ by changing the lockable heap σ accordingly. The rule HEAP-UPD states that given $\ell \mapsto_{\xi} v$, we can temporarily get a HeapLang points-to connective $\ell \mapsto_{\mathsf{HL}} v$ out of the locking heap and update its value.

The environment invariant $\mathsf{env\_inv}(\mathit{env})$ in the definition of $\mathsf{wp}_{\mathsf{mon}}$ ties the contents of the lockable heap to the contents of the environment *env*:

$$\mathsf{env\_inv}(\mathit{env}) \triangleq \exists \sigma\, X.\ \mathsf{is\_set}(\mathit{env}, X) * \mathsf{full\_heap}(\sigma) * (\forall \ell \in X. \exists v.\, \sigma(\ell) = (L, v))$$

The first conjunct states that $X : \wp^{\text{fin}}(\mathit{Loc})$ is a set of locked locations, according to the environment *env*. The second conjunct asserts ownership of the global lockable heap σ. Finally, the last conjunct states that the contents of *env* agrees with the lockable heap: every location that is in X is locked according to σ.

The Unlocking Modality. The unlocking modality is defined in the logic as:

$$\mathbb{U}P \triangleq \exists S.\, (\ast_{(\mathtt{l},\mathtt{v},q)\in S} \mathtt{l} \overset{q}{\mapsto}_L \mathtt{v}) * ((\ast_{(\mathtt{l},\mathtt{v},q)\in S} \mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}) \mathrel{-\!\!*} P)$$

Here S is a finite multiset of tuples containing locations, values, and fractions. The update modality accumulates the locked locations, waiting for them to be unlocked at a sequence point.

4.4 Deriving the λMC Rules

To model weakest preconditions for λMC (Figure 3) we compose the construction we have just defined with the translation of § 2 $\mathsf{wp}\ \mathsf{e}\ \{\Phi\} \triangleq \mathsf{wp}_{\mathsf{mon}}\ [\![\mathsf{e}]\!]\ \{\Phi'\}$. Here, Φ' is the obvious lifting of Φ from λMC values to HeapLang values. Using the rules from Figures 5 and 6 we derive the high-level λMC rules without unfolding the definition of the monadic $\mathsf{wp}_{\mathsf{mon}}$.

Example 4.2. Consider the rule WP-STORE for assignments $\mathsf{e}_1 = \mathsf{e}_2$. Using WP-BIND and WP-PAR, the soundness of WP-STORE can be reduced to verifying the assignment with e_1 being $\mathtt{l}$, e_2 being $\mathtt{v}'$, under the assumption $\mathtt{l} \mapsto_U \mathtt{v}$. We use WP-ATOMIC-ENV to turn our goal into a HeapLang weakest precondition proposition and to gain access an environment *env*, and to the proposition $\mathsf{env\_inv}(\mathit{env})$, from which we extract the lockable heap σ. We then use HEAP-UPD to get access to the underlying HeapLang location and obtain that $\mathtt{l}$ is not locked according to σ. Due to the environment invariant, we obtain that $\mathtt{l}$ is not in *env*, which allows us to prove the `assert` for sequence point violation in the interpretation of the assignment. Finally, we perform the physical update of the location.

5 A Symbolic Executor for λMC

In order to turn our program logic into an automated procedure, it is important to have rules for weakest preconditions that have an algorithmic form. However, the rules for binary operators in our separation logic for λMC do not have such a form. Take for example the rule WP-BIN-OP for binary operators $\mathsf{e}_1 \circledcirc \mathsf{e}_2$. This rule cannot be applied in an algorithmic manner. To use the rule one should supply the postconditions for e_1 and e_2, and frame the resources from the context into two disjoint parts. This is generally impossible to do automatically.

To address this problem, we first describe how the rules for binary operators can be transformed into algorithmic rules by exploiting the notion of *symbolic execution* [5] (§ 5.1). We then show how to implement these algorithmic rules as part of an automated symbolic execution procedure (§ 5.2).

5.1 Rules for Symbolic Execution

We say that we can *symbolically execute* an expression e using a *precondition* P, if we can find a *symbolic execution tuple* $(\mathtt{w}, Q, R)$ consisting of a *return value* $\mathtt{w}$, a *postcondition* Q, and a *frame* R satisfying:

$$P \vdash \mathsf{wp}\ \mathsf{e}\ \{\mathtt{v}.\, \mathtt{v} = \mathtt{w} * Q\} * R$$

This specification is much like that of ordinary symbolic execution in separation logic [5], but there is important difference. Apart from computing the postcondition Q and the return value $\mathtt{w}$, there is also the frame R, which describes the resources that are *not used* for proving $\mathtt{e}$. For instance, if the precondition P is $P' * \mathtt{l} \overset{q}{\mapsto} \mathtt{w}$ and $\mathtt{e}$ is a load operation $*\mathtt{l}$, then we can symbolically execute $\mathtt{e}$ with the postcondition Q being $\mathtt{l} \overset{q/2}{\longmapsto} \mathtt{w}$, and the frame R being $P' * \mathtt{l} \overset{q/2}{\longmapsto} \mathtt{w}$. Clearly, P' is not needed for proving the load, so it can be moved into the frame. More interestingly, since loading the contents of $\mathtt{l}$ requires a read permission $\mathtt{l} \overset{p}{\mapsto} \mathtt{w}$, with $p \in (0, 1]$, we can split the hypothesis $\mathtt{l} \overset{q}{\mapsto} \mathtt{w}$ into two halves and move one into the frame. Below we will see why that matters.

If we can symbolically execute one of the operands of a binary expression $\mathtt{e}_1 \circledcirc \mathtt{e}_2$, say $\mathtt{e}_1$ in P, and find a symbolic execution tuple $(\mathtt{w}_1, Q, R)$, then we can use the following admissible rule:

$$\frac{R \vdash \mathsf{wp}\ \mathtt{e}_2\ \{\mathtt{w}_2.\ Q \mathrel{-\!\!*} \Phi\ (\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)\}}{P \vdash \mathsf{wp}\ (\mathtt{e}_1 \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

This rule has a much more algorithmic flavor than the rule WP-BIN-OP. Applying the above rule now boils down to finding such a tuple $(\mathtt{w}, Q, R)$, instead of having to infer postconditions for both operands, as we need to do to apply WP-BIN-OP.

For instance, given an expression $(*\mathtt{l}) \circledcirc \mathtt{e}_2$ and a precondition $P' * \mathtt{l} \overset{q}{\mapsto} \mathtt{v}$, we can derive the following rule:

$$\frac{P' * \mathtt{l} \overset{q/2}{\longmapsto} \mathtt{v} \vdash \mathsf{wp}\ \mathtt{e}_2 \left\{ \mathtt{w}_2.\ \mathtt{l} \overset{q/2}{\longmapsto} \mathtt{v} \mathrel{-\!\!*} \Phi\ (\mathtt{v}\ [\![\circledcirc]\!]\ \mathtt{w}_2) \right\}}{P' * \mathtt{l} \overset{q}{\mapsto} \mathtt{v} \vdash \mathsf{wp}\ (*\mathtt{l} \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

This rule matches the intuition that only a fraction of the permission $\mathtt{l} \overset{q}{\mapsto} \mathtt{v}$ is needed to prove a load $*\mathtt{l}$, so that the remaining half of the permission can be used to prove the correctness of $\mathtt{e}_2$ (which may contain other loads of $\mathtt{l}$).

5.2 An Algorithm for Symbolic Execution

For an arbitrary expression $\mathtt{e}$ and a proposition P, it is unlikely that one can find such a symbolic execution tuple $(\mathtt{w}, Q, R)$ automatically. However, for a certain class of C expressions that appear in actual programs we can compute a choice of such a tuple. To illustrate our approach, we will define such an algorithm for a small subset $\overline{\mathsf{expr}}$ of C expressions described by the following grammar:

$$\bar{\mathtt{e}} \in \overline{\mathsf{expr}} ::= \mathtt{v} \mid *\bar{\mathtt{e}} \mid \bar{\mathtt{e}}_1 = \bar{\mathtt{e}}_2 \mid \bar{\mathtt{e}}_1 \circledcirc \bar{\mathtt{e}}_2.$$

We keep this subset small to ease presentation. In § 7 we explain how to extend the algorithm to cover the sequenced bind operator $\mathtt{x} \leftarrow \bar{\mathtt{e}}_1\,;\bar{\mathtt{e}}_2$.

Moreover, to implement symbolic execution, we cannot manipulate arbitrary separation logic propositions. We thus restrict to *symbolic heaps* ($m \in \mathsf{sheap}$),

which are defined as finite partial functions $\mathsf{Loc} \xrightarrow{\text{fin}} (\{L, U\} \times (0, 1] \times \mathsf{val})$ representing a collection of points-to propositions:

$$[\![m]\!] \triangleq \mathop{\Large *}_{\substack{\mathtt{l} \in \mathrm{dom}(m) \\ m(\mathtt{l}) = (\xi, \mathtt{q}, \mathtt{v})}} \mathtt{l} \overset{q}{\mapsto}_{\xi} \mathtt{v}.$$

We use the following operations on symbolic heaps:

- $m[\mathtt{l} \mapsto (\xi, q, \mathtt{v})]$ sets the entry $m(\mathtt{l})$ to $(\xi, q, \mathtt{v})$;
- $m \setminus \{\mathtt{l} \mapsto \_\}$ removes the entry $m(\mathtt{l})$ from m;
- $m_1 \sqcup m_2$ merges the symbolic heaps m_1 and m_2 in such a way that for each $\mathtt{l} \in \mathrm{dom}(m_1) \cup \mathrm{dom}(m_2)$, we have:

$$(m_1 \sqcup m_2)(\mathtt{l}) = \begin{cases} m_i(\mathtt{l}) & \text{if } \mathtt{l} \in \mathrm{dom}(m_i) \text{ and } \mathtt{l} \notin \mathrm{dom}(m_j) \\ (\xi \vee \xi', q + q', \mathtt{v}) & \text{if } m_1(\mathtt{l}) = (\xi, q, \mathtt{v}) \text{ and } m_2(\mathtt{l}) = (\xi', q', \_). \end{cases}$$

With this representation of propositions, we define the symbolic execution algorithm as a partial function $\mathsf{forward} : (\mathsf{sheap} \times \mathsf{expr}) \to (\mathsf{val} \times \mathsf{sheap} \times \mathsf{sheap})$, which satisfies the specification stated in § 5.1, *i.e.,* for which the following holds:

Theorem 5.1. *Given an expression* $\mathtt{e}$ *and an symbolic heap* m*, if* $\mathsf{forward}(m, \mathtt{e})$ *returns a tuple* $(\mathtt{w}, m_1^o, m_1)$*, then* $[\![m]\!] \vdash \mathsf{wp}\ \mathtt{e}\ \{\mathtt{v}.\ \mathtt{v} = \mathtt{w} * [\![m_1^o]\!]\} * [\![m_1]\!]$.

The definition of the algorithm is shown in Figure 7. Given a tuple $(m, \mathtt{e})$, a call to $\mathsf{forward}(m, \mathtt{e})$ either returns a tuple $(\mathtt{v}, m^o, m')$ or fails, which either happens when $\mathtt{e} \notin \overline{\mathsf{expr}}$ or when one of intermediate steps of computation fails. In the latter cases, we write $\mathsf{forward}(m, \mathtt{e}) = \bot$.

The algorithm proceeds by case analysis on the expression $\mathtt{e}$. In each case, the expected output is described by the equation $\mathsf{forward}(m, \mathtt{e}) = (\mathtt{v}, m^o, m')$. The results of the intermediate computations appear on separate lines under the clause "**where** ...". If one of the corresponding equations does not hold, *e.g.,* a recursive call fails, then the failure is propagated. Let us now explain the case for the assignment operator.

If $\mathtt{e}$ is an assignment operator $\mathtt{e}_1 = \mathtt{e}_2$, we first evaluate $\mathtt{e}_1$ and then $\mathtt{e}_2$. Fixing the order of symbolic execution from left to right does not compromise the non-determinism underlying the C semantics of binary operators. Indeed, when $\mathsf{forward}(m, \mathtt{e}_1) = (\mathtt{v}_1, m_1^o, m_1)$, we evaluate the expression $\mathtt{e}_2$, using the frame m_1, *i.e.,* only the resources of m that remain after the execution of $\mathtt{e}_1$. When $\mathsf{forward}(m, \mathtt{e}_1) = (\mathtt{l}, m_1^o, m_1)$, with $\mathtt{l} \in \mathsf{Loc}$, and $\mathsf{forward}(m_1, \mathtt{e}_2) = (\mathtt{v}_2, m_2^o, m_2)$, the function $\mathsf{delete\_full\_2}(\mathtt{l}, m_2, m_1^o \sqcup m_2^o)$ checks whether $(m_2 \sqcup m_1^o \sqcup m_2^o)(\mathtt{l})$ contains the write permission $\mathtt{l} \mapsto_U \_$. If this holds, it removes the location $\mathtt{l}$, so that the write permission is now consumed. Finally, we merge $\{\mathtt{l} \mapsto (L, 1, \mathtt{v}_2)\}$ with the output heap m_3^o, so that after assignment, the write permission $\mathtt{l} \mapsto_L \mathtt{v}_2$ is given back in a locked state.

$$\mathsf{forward}(m, \mathtt{v}) \triangleq (\mathtt{v}, \varnothing, m)$$

$$\mathsf{forward}(m, \mathtt{e}_1 \circledcirc \mathtt{e}_2) \triangleq (\mathtt{v}_1 \mathbin{[\![\circledcirc]\!]} \mathtt{v}_2, m_1^o \sqcup m_2^o, m_2)$$
$$\textbf{where}\ (\mathtt{v}_1, m_1^o, m_1) = \mathsf{forward}(m, \mathtt{e}_1)$$
$$(\mathtt{v}_2, m_2^o, m_2) = \mathsf{forward}(m_1, \mathtt{e}_2)$$

$$\mathsf{forward}(m, *\mathtt{e}_1) \triangleq (\mathtt{w}, m_2^o \sqcup \{\mathtt{l} \mapsto (U, q, \mathtt{w})\}, m_2)$$
$$\textbf{where}\quad (\mathtt{l}, m_1^o, m_1) = \mathsf{forward}(m, \mathtt{e}_1) \qquad \text{provided } \mathtt{l} \in \mathsf{Loc}$$
$$(m_2, m_2^o, q, \mathtt{w}) = \mathsf{delete\_frac\_2}(\mathtt{l}, m_1, m_1^o)$$

$$\mathsf{forward}(m, \mathtt{e}_1 = \mathtt{e}_2) \triangleq (\mathtt{v}_2, m_3^o \sqcup \{\mathtt{l} \mapsto (L, 1, \mathtt{v}_2)\}, m_3)$$
$$\textbf{where}\quad (\mathtt{l}, m_1^o, m_1) = \mathsf{forward}(m, \mathtt{e}_1) \qquad \text{provided } \mathtt{l} \in \mathsf{Loc}$$
$$(\mathtt{v}_2, m_2^o, m_2) = \mathsf{forward}(m_1, \mathtt{e}_2)$$
$$(m_3, m_3^o) = \mathsf{delete\_full\_2}(\mathtt{l}, m_2, m_1^o \sqcup m_2^o)$$

$$\mathsf{forward}(m, \mathtt{e}) \triangleq \bot \qquad \text{if } \mathtt{e} \notin \overline{\mathsf{expr}}$$

Auxiliary functions:

$$\mathsf{delete\_frac\_2}(\mathtt{l}, m_1, m_2) \triangleq \begin{cases} (m_1[\mathtt{l} \mapsto (U, q/2, \mathtt{v})], m_2, q/2, \mathtt{v}) & \text{if } m_1(\mathtt{l}) = (U, q, \mathtt{v}) \\ (m_1, m_2[\mathtt{l} \mapsto (U, q/2, \mathtt{v})], q/2, \mathtt{v}) & \text{if } m_1(\mathtt{l}) \neq (U, \_, \_),\ m_2(\mathtt{l}) = (U, q, \mathtt{v}) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{delete\_full\_2}(\mathtt{l}, m_1, m_2) \triangleq (m_1 \setminus \{\mathtt{l} \mapsto \_\}, m_2 \setminus \{\mathtt{l} \mapsto \_\})$$
$$\textbf{where}\ (U, 1, \_) = (m_1 \sqcup m_2)(\mathtt{l})$$

Fig. 7. The definition of the symbolic executor.

6 A Verification Condition Generator for λMC

To establish correctness of programs, we need to prove goals $P \vdash \mathsf{wp}\ \mathtt{e}\ \{\Phi\}$. To prove such a goal, one has to repeatedly apply the rules for weakest preconditions, intertwined with logical reasoning. In this section we will automate this process for λMC by means of a *verification condition generator* (vcgen).

As a first attempt to define a vcgen, one could try to recurse over the expression $\mathtt{e}$ and apply the rules in Figure 3 eagerly. This would turn the goal into a separation logic proposition that subsequently should be solved. However, as we pointed out in §5.1, the resulting separation logic proposition will be very difficult to prove—either interactively or automatically—due to the existentially quantified postconditions that appear because of uses of the rules for binary operators (*e.g.,* WP-BIN-OP). We then proposed alternative rules that avoid the need for existential quantifiers. These rules look like:

$$\frac{R \vdash \mathsf{wp}\ \mathtt{e}_2\ \{\mathtt{v}_2.\ Q \mathrel{-\!\!*} \Phi\ (\mathtt{v}_1 \mathbin{[\![\circledcirc]\!]} \mathtt{v}_2)\}}{P \vdash \mathsf{wp}\ (\mathtt{e}_1 \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

To use this rule, the crux is to symbolically execute e_1 with precondition P into a symbolic execution triple $(\mathtt{v}_1, Q, R)$, which we alluded could be automatically computed by means of the symbolic executor if $\mathsf{e}_1 \in \overline{\mathsf{expr}}$ (§5.2).

We can only use the symbolic executor if P is of the shape $[\![m]\!]$ for a symbolic heap m. However, in actual program verification, the precondition P is hardly ever of that shape. In addition to a series of points-to connectives (as described by a symbolic heap), we may have arbitrary propositions of separation logic, such as pure facts, abstract predicates, nested Hoare triples, Iris ghost state, *etc.* These propositions may be needed to prove intermediate verification conditions, *e.g.,* for function calls. As such, to effectively apply the above rule, we need to separate our precondition P into two parts: a symbolic heap $[\![m]\!]$ and a remainder P'. Assuming $\mathsf{forward}(m, \mathsf{e}_1) = (\mathtt{v}_1, m_1^o, m_1)$, we may then use the following rule:

$$\frac{P' * [\![m_1]\!] \vdash \mathsf{wp}\ \mathsf{e}_2\ \{\mathtt{v}_2.\ [\![m_1^o]\!] \mathrel{-\!\!*} \Phi\ (\mathtt{v}_1\ [\![\circledcirc]\!]\ \mathtt{v}_2)\}}{P' * [\![m]\!] \vdash \mathsf{wp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

It is important to notice that by applying this rule, the remainder P' remains in our precondition as is, but the symbolic heap is changed from $[\![m]\!]$ into $[\![m_1]\!]$, *i.e.,* into the frame that we obtained by symbolically executing e_1.

It should come as no surprise that we can automate this process, by applying rules, such as the one we have given above, recursively, and threading through symbolic heaps. Formally, we do this by defining the vcgen as a total function: $\mathsf{vcg} : (\mathsf{sheap} \times \mathsf{expr} \times (\mathsf{sheap} \to \mathsf{val} \to \mathsf{Prop})) \to \mathsf{Prop}$ where Prop is the type of propositions of our logic. The definition of vcg is given in Figure 8. Before explaining the details, let us state its correctness theorem:

Theorem 6.1. *Given an expression* e*, a symbolic heap* m*, and a postcondition* Φ*, the following statement holds:*

$$\frac{P' \vdash \mathsf{vcg}\,(m, \mathsf{e}, \lambda\, m'\, \mathtt{v}.\ [\![m']\!] \mathrel{-\!\!*} \Phi\ \mathtt{v})}{P' * [\![m]\!] \vdash \mathsf{wp}\ \mathsf{e}\ \{\Phi\}}$$

This theorem reflects the general shape of the rules we previously described. We start off with a goal $P' * [\![m]\!] \vdash \mathsf{wp}\ \mathsf{e}\ \{\Phi\}$, and after using the vcgen, we should prove that the generated goal follows from P'. It is important to note that the continuation in the vcgen is not only parameterized by the return value, but also by a symbolic heap corresponding to the resources that remain. To get these resources back, the vcgen is initiated with the continuation $\lambda\, m'\, \mathtt{v}.\ [\![m']\!] \mathrel{-\!\!*} \Phi\ \mathtt{v}$.

Most clauses of the definition of the vcgen (Figure 8) follow the approach we described so far. For unary expressions like load we generate a condition that corresponds to the weakest precondition rule. For binary expressions, we symbolically execute either operand, and proceed recursively in the other. There are a number of important bells and whistles that we will discuss now.

Sequencing. In the case of sequenced binds $\mathtt{x} \leftarrow \mathsf{e}_1\,;\mathsf{e}_2$, we recursively compute the verification condition for e_1 with the continuation:

$$\lambda\, m'\, \mathtt{v}.\ \mathbb{U}\ (\mathsf{vcg}\,(\mathsf{unlock}\,(m'), \mathsf{e}_2\,[\mathtt{v}/\mathtt{x}], \mathcal{K}))\,.$$

$$\mathsf{vcg}(m, \mathtt{v}, \mathcal{K}) \triangleq \mathcal{K}\ m\ \mathtt{v}$$

$$\mathsf{vcg}(m, \mathsf{e}_1 \circledcirc \mathsf{e}_2, \mathcal{K}) \triangleq \begin{cases} \mathsf{vcg}\left(m_2, \mathsf{e}_2, \lambda\, m'\ \mathtt{v}_2.\ \mathcal{K}\ (m' \sqcup m^o)\ (\mathtt{v}_1 \circledcirc \mathtt{v}_2)\right) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = (\mathtt{v}_1, m^o, m_2) \\ \mathsf{vcg}\left(m_1, \mathsf{e}_1, \lambda\, m'\ \mathtt{v}_1.\ \mathcal{K}\ (m' \sqcup m^o)\ (\mathtt{v}_1 \circledcirc \mathtt{v}_2)\right) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = \bot \text{ and } \mathsf{forward}(m, \mathsf{e}_2) = (\mathtt{v}_2, m^o, m_1) \\ [\![m]\!] \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\mathcal{K}_{\mathtt{ret}}\} & \text{otherwise} \end{cases}$$

$$\mathsf{vcg}(m, *\mathsf{e}, \mathcal{K}) \triangleq \mathsf{vcg}(m, \mathsf{e}, \mathcal{K}')$$

$$\textbf{with } \mathcal{K}' \triangleq \lambda\, m\, \mathtt{l}.\ \begin{cases} \mathcal{K}\ m\ \mathtt{w} & \text{if } \mathtt{l} \in \mathsf{Loc} \text{ and } m(\mathtt{l}) = (U, q, \mathtt{w}) \\ [\![m]\!] \mathrel{-\!\!*} \exists \mathtt{w}\ q.\ \mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} * (\mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} \mathrel{-\!\!*} \mathcal{K}_{\mathtt{ret}}\ \mathtt{w}) & \text{otherwise} \end{cases}$$

$$\mathsf{vcg}(m, \mathsf{e}_1 = \mathsf{e}_2, \mathcal{K}) \triangleq \begin{cases} \mathsf{vcg}\left(m_2, \mathsf{e}_2, \lambda\, m'\ \mathtt{v}.\ \mathcal{K}'\ (m' \sqcup m^o)(\mathtt{l}, \mathtt{v})\right) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = (\mathtt{l}, m^o, m_2) \\ \mathsf{vcg}\left(m_1, \mathsf{e}_1, \lambda\, m'\ \mathtt{l}.\ \mathcal{K}'\ (m' \sqcup m^o)(\mathtt{l}, \mathtt{v})\right) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = \bot \text{ and } \mathsf{forward}(m, \mathsf{e}_2) = (\mathtt{v}, m^o, m_1) \\ [\![m]\!] \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\mathcal{K}_{\mathtt{ret}}\} & \text{otherwise} \end{cases}$$

$$\textbf{with } \mathcal{K}' \triangleq \lambda\, m\ (\mathtt{l}, \mathtt{v}).\ \begin{cases} \mathcal{K}\ (m' \sqcup \{\mathtt{l} \mapsto (L, 1, \mathtt{v})\})\ \mathtt{v} & \text{if } \mathtt{l} \in \mathsf{Loc} \text{ and } \mathsf{delete\_full}(\mathtt{l}, m) = m' \\ [\![m]\!] \mathrel{-\!\!*} \exists \mathtt{w}.\ \mathtt{l} \mapsto_U \mathtt{w} * (\mathtt{l} \mapsto_L \mathtt{v} \mathrel{-\!\!*} \mathcal{K}_{\mathtt{ret}}\ \mathtt{v}) & \text{otherwise} \end{cases}$$

$$\mathsf{vcg}(m, \mathtt{x} \leftarrow \mathsf{e}_1\,;\mathsf{e}_2, \mathcal{K}) \triangleq \mathsf{vcg}\left(m, \mathsf{e}_1, \lambda\, m'\ \mathtt{v}.\ \mathbb{U}\ \left(\mathsf{vcg}(\mathsf{unlock}(m'), \mathsf{e}_2[\mathtt{v}/\mathtt{x}], \mathcal{K})\right)\right)$$

Auxiliary functions:

$$\mathcal{K}_{\mathtt{ret}} : \mathsf{val} \to \mathsf{Prop} \triangleq \lambda\, \mathtt{w}.\ (\exists m'.\ [\![m']\!] * \mathcal{K}\ m'\ \mathtt{w}) \qquad \mathsf{unlock}(m) \triangleq \bigsqcup_{\substack{\mathtt{l} \in \mathrm{dom}(m) \\ m(\mathtt{l}) = (\_, \mathtt{q}, \mathtt{v})}} \{\mathtt{l} \mapsto (U, q, \mathtt{v})\}$$

Fig. 8. Selected cases of the verification condition generator.

Due to a sequence point, all locations modified by e_1 will be in the unlocked state after it is finished executing. Therefore, in the recursive call to e_2 we unlock all locations in the symbolic heap (*c.f.* $\mathsf{unlock}(m')$), and we include a $\mathbb{U}$ modality in the continuation. The $\mathbb{U}$ modality is crucial so that the resources that are not given to the vcgen (the remainder P' in Theorem 6.1) can also be unlocked.

Handling Failure. In the case of binary operators $\mathsf{e}_1 \circledcirc \mathsf{e}_2$, it could be that the symbolic executor fails on both e_1 and e_2, because neither of the arguments were of the right shape (*i.e.,* not an element of $\overline{\mathsf{expr}}$), or the required resources were not present in the symbolic heap. In this case the vcgen generates the goal of the form $[\![m]\!] \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\mathcal{K}_{\mathtt{ret}}\}$ where $\mathcal{K}_{\mathtt{ret}} \triangleq \lambda\, \mathtt{w}.\ \exists m'.\ [\![m']\!] * \mathcal{K}\ m'\ \mathtt{w}$.

What appears here is that the current symbolic heap $[\![m]\!]$ is given back to the user, which they can use to prove the weakest precondition of $e_1 \circledcirc e_2$ by hand. Through the postcondition $\exists m'.\, [\![m']\!] * \mathcal{K}\ m'\ \mathtt{w}$ the user can resume the vcgen, by choosing a new symbolic heap m' and invoking the continuation $\mathcal{K}\, m'\, \mathtt{w}$.

For assignments $\mathtt{e_1} = \mathtt{e_2}$ we have a similar situation. Symbolic execution of both $\mathtt{e_1}$ and $\mathtt{e_2}$ may fail, and then we generate a goal similar to the one for binary operators. If the location $\mathtt{l}$ that we wish to assign to is not in the symbolic heap, we use the continuation $[\![m]\!] \mathrel{-\!\!*} \exists \mathtt{w}.\, \mathtt{l} \mapsto_U \mathtt{w} * (\mathtt{l} \mapsto_L \mathtt{v} \mathrel{-\!\!*} \mathcal{K}_{\mathtt{ret}}\ \mathtt{v})$. As before, the user gets back the current symbolic heap $[\![m]\!]$, and could resume the vcgen through the postcondition $\mathcal{K}_{\mathtt{ret}}\ \mathtt{v}$ by picking a new symbolic heap.

7 Discussion

Extensions of the Language. The memory model that we have presented in this paper was purposely oversimplified. In Coq, the memory model for λMC additionally supports mutable local variables, arrays, and pointer arithmetic. Adding support for these features was relatively easy and required only local changes to the definitional semantics and the separation logic.

For implementing mutable local variables, we tag each location with a Boolean that keeps track of whether it is an allocated or a local variable. That way, we can forbid deallocating local variables using the $\mathtt{free}(-)$ operator.

Our extended memory model is block/offset-based like CompCert's memory model [38]. Pointers are not simply represented as locations, but as pairs (ℓ, i), where ℓ is a HeapLang reference to a memory block containing a list of values, and i is an offset into that block. The points-to connectives of our separation logic then correspondingly range over block/offset-based pointers.

Symbolic Execution of Sequence Points. We adapt our forward algorithm to handle sequenced bind operators $\mathtt{x} \leftarrow \mathtt{e_1}\,;\mathtt{e_2}$. The subtlety lies in supporting nested sequenced binds. For example, in an expression $(\mathtt{x} \leftarrow \mathtt{e_1}\,;\mathtt{e_2}) + \mathtt{e_3}$ the postcondition of $\mathtt{e_1}$ can be used (along with the frame) for the symbolic execution of $\mathtt{e_2}$, but it cannot be used for the symbolic execution of $\mathtt{e_3}$. In order to solve this, our forward algorithm takes a *stack* of symbolic heaps as an input, and returns a *stack* of symbolic heaps (of the same length) as a frame. All the cases shown in Figure 7 are easily adapted w.r.t. this modification, and the following definition captures the case for the sequence point bind:

$$
\begin{aligned}
&\mathsf{forward}\,(\vec{m}, \mathtt{x} \leftarrow \mathtt{e_1}\,;\mathtt{e_2}) \triangleq (\mathtt{v_2}, m_2^o \sqcup m', \vec{m_2}) \\
&\quad \textbf{where}\ \ (\mathtt{v_1}, m_1^o, \vec{m}_1) \qquad\quad = \mathsf{forward}\,(\vec{m}, \mathtt{e_1}) \\
&\qquad\qquad (\mathtt{v_2}, m_2^o, m' :: \vec{m}_2) = \mathsf{forward}\,(\mathsf{unlock}\,(m_1^o) :: \vec{m}_1, \mathtt{e_2}\,[\mathtt{v_1/x}])
\end{aligned}
$$

Shared Resource Invariants. As in Krebbers's logic [29], the rules for binary operators in Figure 3 require the resources to be separated into disjoint parts for the subexpressions. If both sides of a binary operator are function calls, then they

can only share read permissions despite that both function calls are executed atomically. Following Krebbers, we address this limitation by adding a shared resource invariant R to our weakest preconditions and add the following rules:

$$\frac{R_1 \qquad \mathsf{wp}_{R_1 * R_2}\, \mathtt{e}\, \{\mathtt{v}.\, R_1 \mathrel{-\!\!*} \Phi\, \mathtt{v}\}}{\mathsf{wp}_{R_2}\, \mathtt{e}\, \{\Phi\}} \qquad \frac{\mathtt{f(x)\{e\}}\text{ defined} \qquad R \mathrel{-\!\!*} \mathbb{U}(\mathsf{wp}_{\mathsf{True}}\, \mathtt{e}[\mathtt{x}/\mathtt{v}]\, \{\mathtt{w}.\, R * \Phi\, \mathtt{w}\})}{\mathsf{wp}_R\, \mathtt{f(v)}\, \{\Phi\}}$$

To temporarily transfer resources into the invariant, one can use the first rule. Because function calls are not interleaved, one can use the last rule to gain access to the shared resource invariant for the duration of the function call.

Our handling of shared resource invariants generalizes the treatment by Krebbers: using custom ghost state in Iris we can endow the resource invariant with a protocol. This allows us to verify examples that were previously impossible [29]:

```
int f(int *p, int y) { return (*p = y); }
int main() { int x; f(&x, 3) + f(&x, 4); return x; }
```

Krebbers could only prove that `main` returns 0, 3 or 4, whereas we can prove it returns 3 or 4 by combining resource invariants with Iris's ghost state.

Implementation in Coq. In the Coq development [18] we have:

- Defined λMC with the extensions described above, as well as the monadic combinators, as a shallow embedding on top of Iris's HeapLang [25,21].
- Modeled the separation logic for λMC and the monadic combinators as a shallow embedding on top of the Iris's program logic for HeapLang.
- Implemented the symbolic executor and vcgen as computable Coq functions, and proved their soundness w.r.t. our separation logic.
- Turned the verification condition generator into a tactic that integrates into the Iris Proof Mode/MoSeL framework [34,32].

This last point allowed us to leverage the existing machinery for separation logic proofs in Coq. Firstly, we get basic building blocks for implementing the vcgen tactic for free. Secondly, when the vcgen is unable to solve the goal, one can use the Iris Proof Mode/MoSeL tactics to help out in a convenient manner.

To implement the symbolic executor and vcgen, we had to reify the terms and values of λMC. To see why reification is needed, consider the data type for symbolic heaps, which uses locations as keys. In proofs, those locations appear as universally quantified variables. To compute using these, we need to reify them into some symbolic representation. We have implemented the reification mechanism using type classes, following Spitters and van der Weegen [47].

With all the mechanics in place, our vcgen is able to significantly aid us. Consider the following program that copies the contents of one array into another:

```
int arraycopy(int *p, int *q, int n) {
  int pend = p + n;
  while (p < pend) { *(p++) = *(q++); }
}
```

We proved $\{\mathtt{p} \mapsto \vec{x} * \mathtt{q} \mapsto \vec{y} * (|\vec{x}| = |\vec{y}| = n)\}\ \mathtt{arraycopy(p, q, n)}\ \{\mathtt{p} \mapsto \vec{y} * \mathtt{q} \mapsto \vec{y}\}$ in 11 lines of Coq code. The vcgen can automatically process the program up until the while loop. At that point, the user has to manually perform an induction on the array, providing a suitable induction hypothesis. The vcgen is then able to discharge the base case automatically. In the inductive case, it will automatically process the program until the next iteration of the while loop, where the user has to apply the induction hypothesis.

8 Related Work

C Semantics. There has been a considerable body of work on formal semantics for the C language, including several large projects that aimed to formalize substantial subsets of C [44,37,17,20,41,30], and projects that focused on specific aspects like its memory model [41,28,31,38,13,27,10,40], weak memory concurrency [4,36,43], non-local control flow [35], verified compilation [37,48], *etc.*

The focus of this paper—non-determinism in C expressions—has been treated formally a number of times, notably by Norrish [44], Ellison and Rosu [17], Krebbers [31], and Memarian *et al.* [41]. The first three have in common that they model the sequence point restriction by keeping track of the locations that have been written to. The treatment of sequence points in our definitional semantics is closely inspired by the work of Ellison and Rosu [17], which resembles closely what is in the C standard. Krebbers [31] used a more restrictive version of the semantics by Ellison and Rosu—he assigned undefined behavior in some corner cases to ease the soundness theorem of his logic. We directly proved soundness of the logic w.r.t. the more faithful model by Ellison and Rosu.

Memarian *et al.* [41] give a semantics to C by elaboration into a language they call Core. Unspecified evaluation order in Core is modeled using an unseq operation, which is similar to our $||_{\mathsf{HL}}$ operation. Compared to our translation, Core is much closer to C (it has function calls, memory operations, *etc.* as primitives, while we model them with monadic combinators), and supports concurrency.

Reasoning Tools and Program Logics for C. Apart from formalizing the semantics of C, there have been many efforts to create reasoning tools for the C language in one way or another. There are standalone tools, like VeriFast [23], VCC [12], and the Jessie plugin of Frama-C [42], and there are tools built on top of general purpose proof assistants like VST [1,10] in Coq, or AutoCorres [19] in Isabelle/HOL. Although, admittedly, all of these tools cover larger subsets of C than we do, as far as we know, they all ignore non-determinism in expressions.

There are a few exceptions. Norrish proved confluence for a certain class of C expressions [45]. Such a confluence result may be used to justify proofs in a tool that does not have an underlying non-deterministic semantics.

Another exception is the separation logic for non-determinism in C by Krebbers [29]. Our work is inspired by his, but there are several notable differences:

– We have proved soundness with respect to a definitional semantics for a subset of C. We believe that this approach is more modular, since the semantics can be specified at a higher level of abstraction.

- We have built our logic on top of the Iris framework. This makes the development more modular (since we can use all the features as well as the Coq infrastructure of Iris) and more expressive (as shown in § 7).
- There was no automation like our vcgen, so one had to subdivide resources between subexpressions manually all the time. Also, there was not even tactical support for carrying out proofs manually. Our logic is redesigned to get such support from the Iris Proof Mode/MoSeL framework.

To handle missing features of C as part of our vcgen, we plan to explore approaches by other verification projects in proof assistants. A notable example of such a project is VST, which supports machine arithmetic [16] and data types like structs and unions [10] as part of its tactics for symbolic execution.

Separation Logic and Symbolic Execution. In their seminal work, Berdine *et al.* [5] demonstrate the application of symbolic execution to automated reasoning in separation logic. In their setting, frame inference is used to perform symbolic execution of function calls. The frame has to be computed when the call site has more resources than needed to invoke a function. In our setting we compute frames for subexpressions, which, unlike functions, do not have predefined specifications. Due to that, we have to perform frame inference simultaneously with symbolic execution. The symbolic execution algorithm of Berdine *et al.* can handle inductive predicates, and can be extended with shape analysis [15]. We do not support such features, and leave them to future work.

Caper [14] is a tool for automated reasoning in concurrent separation logic, and it also deals with non-determinism, although the nature of non-determinism in Caper is different. Non-determinism in Caper arises due to branching on unknown conditionals and due to multiple possible ways to apply ghost state related rules (rules pertaining to abstract regions and guards). The former cause is tackled by considering sets of symbolic execution traces, and the latter is resolved by employing heuristics based on bi-abduction [9]. Applications of abductive reasoning to our approach to symbolic execution are left for future work.

Recently, Bannister *et al.* [3,2] proposed a new separation logic connective for performing forwards reasoning whilst avoiding frame inference. This approach, however, is aimed at sequential deterministic programs, focusing on a notion of partial correctness that allows for failed executions. Another approach to verification of sequential stateful programs is based on characteristic formulae [11]. A stateful program is transformed into a higher-order logic predicate, implicitly encoding the frame rule. The resulting formula is then proved by a user in Coq.

When implementing a vcgen in a proof assistant (see *e.g.,* [39,10]) it is common to let the vcgen return a new goal when it gets stuck, from which the user can help out and call back the vcgen. The novelty of our work is that this approach is applied to operations that are called in parallel.

Acknowledgments. We are grateful to Gregory Malecha and the anonymous reviewers and for their comments and suggestions. This work was supported by the Netherlands Organisation for Scientific Research (NWO), project numbers STW.14319 (first and second author) and 016.Veni.192.259 (third author).

References

1. Appel, A.W. (ed.): Program Logics for Certified Compilers. Cambridge University Press (2014)
2. Bannister, C., Höfner, P.: False Failure: Creating Failure Models for Separation Logic. In: RAMiCS. LNCS, vol. 11194, pp. 263–279 (2018)
3. Bannister, C., Höfner, P., Klein, G.: Backwards and Forwards with Separation Logic. In: ITP. LNCS, vol. 10895, pp. 68–87 (2018)
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ Concurrency. In: POPL. pp. 55–66 (2011)
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: APLAS. LNCS, vol. 3780, pp. 52–68 (2005)
6. Birkedal, L., Bizjak, A.: Lecture Notes on Iris: Higher-Order Concurrent Separation Logic (August 2018), https://iris-project.org/tutorial-material.html
7. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission Accounting in Separation Logic. In: POPL. pp. 259–270 (2005)
8. Boyland, J.: Checking interference with fractional permissions. In: SAS. LNCS, vol. 2694, pp. 55–72 (2003)
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. J. ACM **58**(6), 26:1–26:66 (2011)
10. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. JAR **61**(1-4), 367–422 (2018)
11. Charguéraud, A.: Characteristic Formulae for the Verification of Imperative Programs. SIGPLAN Not. **46**(9), 418–430 (2011)
12. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOLs. LNCS, vol. 5674, pp. 23–42 (2009)
13. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A Precise Yet Efficient Memory Model For C. ENTCS **254**, 85–103 (2009)
14. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J., Birkedal, L.: Caper - Automatic Verification for Fine-Grained Concurrency. In: ESOP. LNCS, vol. 10201, pp. 420–447 (2017)
15. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: TACAS. LNCS, vol. 3920, pp. 287–302 (2006)
16. Dodds, J., Appel, A.W.: Mostly Sound Type System Improves a Foundational Program Verifier. In: CPP. LNCS, vol. 8307, pp. 17–32 (2013)
17. Ellison, C., Rosu, G.: An Executable Formal Semantics of C with Applications. In: POPL. pp. 533–544 (2012)
18. Frumin, D., Gondelman, L., Krebbers, R.: Semi-Automated Reasoning About Non-Determinism in C Expressions: Coq Development (February 2019), https://cs.ru.nl/~dfrumin/wpc/
19. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In: PLDI. pp. 429–439 (2014)
20. Hathhorn, C., Ellison, C., Roşu, G.: Defining the Undefinedness of C. In: PLDI. pp. 336–345 (2015)
21. Iris: Iris Project (November 2018), https://iris-project.org/
22. ISO: ISO/IEC 9899-2011: Programming Languages – C. ISO Working Group 14 (2012)

23. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: APLAS. LNCS, vol. 6461, pp. 304–311 (2010)
24. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order Ghost State. In: ICFP. pp. 256–269 (2016)
25. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris From The Ground Up: A Modular Foundation for Higher-order Concurrent Separation Logic. Journal of Functional Programming **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
26. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent reasoning. In: POPL. pp. 637–650 (2015)
27. Kang, J., Hur, C., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A Formal C Memory Model Supporting Integer-Pointer Casts. In: POPL. pp. 326–335 (2015)
28. Krebbers, R.: Aliasing Restrictions of C11 Formalized in Coq. In: CPP. LNCS, vol. 8307 (2013)
29. Krebbers, R.: An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In: POPL. pp. 101–112 (2014)
30. Krebbers, R.: The C Standard Formalized in Coq. Ph.D. thesis, Radboud University Nijmegen (2015)
31. Krebbers, R.: A formal C memory model for separation logic. JAR **57**(4), 319–387 (2016)
32. Krebbers, R., Jourdan, J., Jung, R., Tassarotti, J., Kaiser, J., Timany, A., Charguéraud, A., Dreyer, D.: MoSeL: a General, Extensible Modal Framework for Interactive Proofs in separation logic. PACMPL **2**(ICFP), 77:1–77:30 (2018)
33. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J., Dreyer, D., Birkedal, L.: The Essence of Higher-order Concurrent Separation Logic. In: ESOP. LNCS, vol. 10201, pp. 696–723 (2017)
34. Krebbers, R., Timany, A., Birkedal, L.: Interactive Proofs in Higher-order Concurrent Separation Logic. In: POPL. pp. 205–217 (2017)
35. Krebbers, R., Wiedijk, F.: Separation Logic for Non-local Control Flow and Block Scope Variables. In: FoSSaCS. LNCS, vol. 7794, pp. 257–272 (2013)
36. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing Sequential Consistency in C/C++11. In: PLDI. pp. 618–632 (2017)
37. Leroy, X.: Formal Verification of a Realistic Compiler. CACM **52**(7), 107–115 (2009)
38. Leroy, X., Blazy, S.: Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. JAR **41**(1), 1–31 (2008)
39. Malecha, G.: Extensible Proof Engineering in Intensional Type Theory. Ph.D. thesis, Harvard University (2014)
40. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring C semantics and pointer provenance. PACMPL **3**(POPL), 67:1–67:32 (2019)
41. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the Depths of C: Elaborating the De Facto Standards. In: PLDI. pp. 1–15 (2016)
42. Moy, Y., Marché, C.: The Jessie Plugin for Deduction Verification in Frama-C, Tutorial and Reference Manual (2011)
43. Nienhuis, K., Memarian, K., Sewell, P.: An Operational Semantics for C/C++11 Concurrency. In: OOPSLA. pp. 111–128 (2016)

44. Norrish, M.: C Formalised in HOL. Ph.D. thesis, University of Cambridge (1998)
45. Norrish, M.: Deterministic Expressions in C. In: ESOP. LNCS, vol. 1576, pp. 147–161 (1999)
46. OHearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science **375**(1), 271 – 307 (2007), festschrift for John C. Reynoldss 70th birthday
47. Spitters, B., Van der Weegen, E.: Type Classes for Mathematics in Type Theory. Mathematical Structures in Computer Science **21**(4), 795–825 (2011)
48. Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional CompCert. In: POPL. pp. 275–287 (2015)